SAND REPORT

SAND2003-0249
Unlimited Release
Printed January 2003

SIERRA Framework Version 3: Transfer Services Design and Use

James R. Stewart

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

SAND2003-0249
Unlimited Release
Printed January 2003

SIERRA Framework Version 3: Transfer Services Design and Use

James R. Stewart
Production Computing/SIERRA Architecture Department
Engineering Sciences Center
Sandia National Laboratories
Box 5800
Albuquerque, NM 87185-0826

Abstract

This paper presents a description of the SIERRA Framework Version 3 parallel transfer operators. The high-level design including object interrelationships, as well as requirements for their use, is discussed. Transfer operators are used for moving field data from one computational mesh to another. The need for this service spans many different applications. The most common application is to enable loose coupling of multiple physics modules, such as for the coupling of a quasi-statics analysis with a thermal analysis. The SIERRA transfer operators support the transfer of nodal and element fields between meshes of different, arbitrary parallel decompositions. Also supplied are "copy" transfer operators for efficient transfer of fields between identical meshes. A "copy" transfer operator is also implemented for constraint objects. Each of these transfer operators is described. Also, two different parallel algorithms are presented for handling the geometric misalignment between different parallel-distributed meshes.

Acknowledgement

The author would like to thank Jason Hales and Jonathan Rath for reviewing the manuscript, and Rhonda Reinert of Technically Write, Inc. for providing substantial editorial assistance.

Contents

1 **Introduction** **7**

1.1 Concepts and Capabilities 7

1.2 Document Organization 8

2 **Transfer Subsystem Overview** **9**

2.1 Transfer Design 9

2.2 Mapping to SIERRA Classes 10

2.3 Intraprocedural Versus Interprocedural Transfers 12

3 **Transfer Operators** **14**

3.1 Nodal Copy 15

3.2 Element Copy 16

3.3 Constraint-Object Copy 17

3.4 Nodal Interpolation 17

3.5 Element-Centroid Least-Squares Projection 20

4 **Parallel Algorithms for Transfers** **26**

4.1 Multiple Concurrent Domain Decompositions 26

4.2 Parallel Algorithm 1 28

4.3 Parallel Algorithm 2 29

4.4 Trade-offs Between Parallel Algorithms 1 and 2 31

5 **Summary and Future Work** **32**

Figures

2.1 Transfer subsystem object model. 9

2.2 Usage of intraprocedural transfers: 12

2.3 Usage of interprocedural transfers. 13

3.1 Illustration of a solvable patch and an unsolvable patch. 22

3.2 Ghosting of mesh objects. 23

4.1 Illustration of multiple concurrent domain decompositions. 28

4.2 Design of parallel algorithm 1 for transfer operators. 29

4.3 Design of parallel algorithm 2 for transfer operators. 31

Tables

2.1 Mapping of Transfer Objects to SIERRA Framework Version 3 C++ classes 11
2.2 Important SIERRA Framework Version 3 Classes Used by Transfer Subsystem 11

SIERRA Framework Version 3: Transfer Services Design and Use

1 Introduction

This paper presents important features of the SIERRA Framework Version 3 transfer subsystem. The high-level design of the subsystem is given, along with descriptions of the transfer operators and their parallel algorithms. The operators are designed to work in a fully distributed parallel environment. In addition to the subsystem design, we also focus on how an application developer constructs and uses the transfer operators. Although this paper does not provide an Application Programming Interface (API), mappings of functionality to specific SIERRA C++ classes are provided. It is recommended that application developers consult the appropriate class header files to obtain the API.

1.1 Concepts and Capabilities

Transfer operators can be thought of herein as collections of (C++) transfer objects working together to move data from one computational mesh to another computational mesh. In version 3 of the SIERRA framework, data is thus moved from one SIERRA region to another SIERRA region, given that there is a one-to-one correlation between a region and a mesh. The two regions may or may not belong to the same SIERRA procedure. The terms *intraprocedural transfer* and *interprocedural transfer* are used in this paper to describe these two cases, respectively. Definitions of the SIERRA region and procedure, along with other useful SIERRA concepts and objects, are described in Ref. 1.

Transfer operators have many applications. First and foremost, they are useful for "loose" coupling of different physics codes. They are also used in *a posteriori* error estimators for transferring residuals and other quantities needed for solving the local problems and for computing inner products. For problems involving large deformations such as penetration problems, transfer operators may be used for moving the solution and state variables to a regenerated adaptive mesh for a restart.

The computational meshes involved in the data transfer can have different, arbitrary parallel decompositions. This is referred to as the different-mesh case. If it is known *a priori* that these meshes are identical (i.e., their nodes and elements have the same global numbering and parallel processor distribution), then much more efficient transfer operators can be developed. This is referred to as the same-mesh case.

The SIERRA Framework Version 3 transfer operators handle three types of variables: nodal (or node) variables, element variables, and field variables registered on constraint objects. For the same-mesh case, there are very efficient nodal, element, and constraint-object "copy" transfer operators. For the different-mesh case, there are transfer operators only for nodal and element variables. The element variables (for the different-mesh case) must be piecewise-constant fields so that these variables can be represented by a single value at the centroid of the element. The (serial) interpolation algorithms for the different-mesh case mirror those found in MAPVAR [2]. Nodal variables are interpolated using the finite-element shape functions, while element variables are interpolated from a trilinear polynomial constructed using a least-squares element-patch projection (in two dimensions, the interpolating polynomial is bilinear; in one dimension, it is linear).

Remark

The element transfer operators are designed to also work with face and edge variables. In this context we speak of face or edge "elements," which can be used, for example, for the transfer of surface fields in three and two dimensions, respectively. However, the element transfer operators have not been tested for this purpose.

For the different-mesh case, the transfer operators consist of two main components: search and interpolation. The interpolation methods used for nodal and element variables were mentioned above. In a distributed parallel environment, as opposed to a shared-memory parallel environment or a serial environment, the search becomes more critical and is very important to the successful execution of the transfer. The search is broken down into two parts—parallel search and local search. The parallel search uses a Recursive Coordinate Bisection (RCB) decomposition to handle the geometric misalignment of the meshes across the processors. The local search uses the point-in-box method [3], which is the same search method used in MAPVAR [2]. The same-mesh "copy" transfer operators achieve their efficiency because the search step is entirely avoided, and the interpolation step is trivial (it is just a data copy).

1.2 Document Organization

In Section 2 we build on the concepts and capabilities introduced above, and provide an overview of the high-level subsystem design. The overview includes a presentation of the object model, a description of how the transfer objects are mapped to C++ classes used in SIERRA, and a discussion of how interprocedural and intraprocedural transfers are used by an application. In Section 3 we present a detailed discussion on the interpolation algorithms used in the transfer operators, including information needed by an application for their usage. Specifically, the interpolation methods are the *copy* and *finite-element shape-function interpolation* algorithms for nodal variables, the *copy* and *least-squares element-patch projection* algorithms for element variables, and the *copy* algorithm for constraint-object variables. As mentioned previously, the parallel aspects of the transfer operators introduce complexity. This complexity leads to certain choices about how the transfers can be carried out in parallel. In Section 4 we present two different parallel algorithms and discuss their trade-offs. A summary and a discussion of possible future work is given in Section 5.

2 Transfer Subsystem Overview

This section presents the object design of the transfer subsystem and provides the relevant C++ classes used in SIERRA Framework Version 3 for implementing the design. We also discuss the usage of *interprocedural* and *intraprocedural* transfers by an application.

2.1 Transfer Design

The transfer subsystem design includes a transfer object as well as various auxilliary objects. The design is shown in Fig. 2.1.

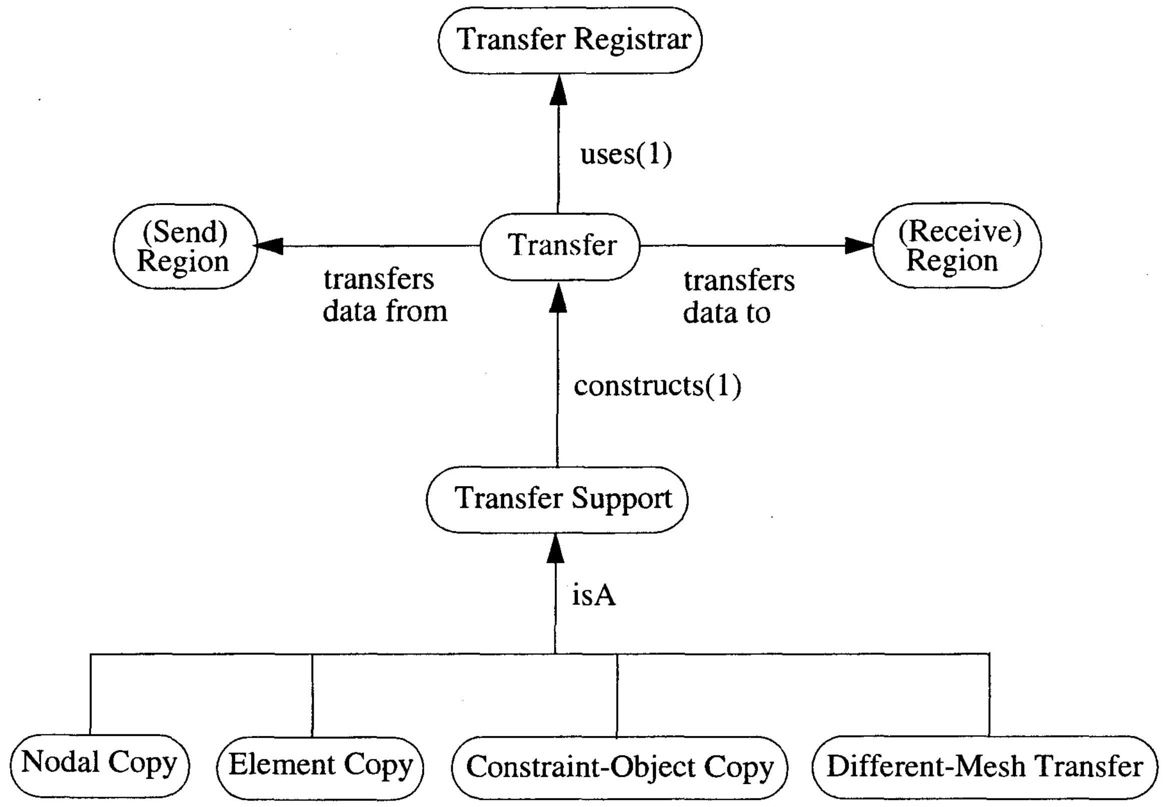

Figure 2.1. Transfer subsystem object model.

The purpose of a transfer operator is to transfer field data, i.e., variables registered on mesh objects, from a send region to a receive region. The variables to be transferred are specified in the transfer registrar. Except for the "copy" transfer operators (nodal copy, element copy, and constraint-object copy), the send region and receive region may have arbitrarily different meshes, implying that these meshes may have arbitrarily different distributions among the processors. The parallel implementation of a transfer operator is completely hidden from the application developer. All transfer operators have the same user interface except possibly for construction and initialization. The "copy" transfer operators do not require an initialization function, while the different-mesh transfer operators require an initialization function as well as additional algorithm control data.

Remark

The "copy" transfer operators require *exactly* the same meshes, with identical parallel decompositions, for both the send and receive regions.

Any number of transfer operators can be implemented. In general, different operators may be implemented as different derived classes from the transfer support base class. Alternatively, multiple operators may be embedded in the same derived transfer support class. The latter design is used for the two different-mesh transfer operators implemented in SIERRA Framework Version 3. These two operators are *nodal interpolation* and *element-centroid least-squares projection*. Specification of the appropriate transfer operator is done through the setting of control data. This aspect of operator usage is explained in more detail in Section 3.4 and Section 3.5. The reason for sharing a single derived transfer support class is that this class sets up and executes the geometric search, including both the local search and the parallel search. The search constitutes a large part of the transfer operator, and the basic steps are the same for both the nodal and element cases. As mentioned in Section 1, we use the point-in-box method [3] for the local search, and an RCB decomposition (via Zoltan [4]) for the parallel search.

To use a transfer operator, the application developer must first construct the transfer object and one or more transfer registrar objects. These steps are usually done inside a parser handler function. There is one registrar for each variable that is transferred. The registrar holds the names and states of the send and receive variables (i.e., variables registered in the send and receive regions; see Fig. 2.1). The state information makes it possible, for example, to transfer a variable from the "new" state in the send mesh to the "old" state in the receive mesh. In general, any combination is possible. The registrar also holds the number of nesting levels of the variables. Currently, only the element copy transfer operator provides support for nested variables; however, support for nesting in the other transfer operators can easily be added.

For intraprocedural transfers, the application developer is responsible for calling the transfer operators from one or more mechanics algorithms. Calls to interprocedural transfer operators that involve two sequentially executing procedures are handled by the SIERRA framework. In general, no additional code needs to be written by the application developer.

2.2 Mapping to SIERRA Classes

The objects shown in Fig. 2.1 correspond to C++ classes in SIERRA Framework Version 3. The class names are given in Table 2.1. In Table 2.2, we provide the important framework classes used by the transfer subsystem. (The `Xfer_ElemCentroidLinear` class shown in Table 2.2 is actually part of the transfer subsystem; however, it is derived from a framework class and *used by* the element-centroid least-squares projection transfer operator, as indicated in the "Notes" column.) It is important to note that in future versions of SIERRA the design and implementation may evolve or change completely, although the underlying transfer operators will likely remain. For example, the implementation of the local search is currently done in the `Fmwk_LocalSearch` class (see Table 2.2). The interface to this class is such that the point-in-box method could be replaced by some other local search method without affecting any other part of the overall transfer operator.

Table 2.1. Mapping of Transfer Objects to SIERRA Framework Version 3 C++ classes

Transfer Object	SIERRA Class
Transfer	`Xfer_Transfer`
Transfer Registrar	`Xfer_TransferRegistrar`
Transfer Support	`Xfer_Transfer::Support`
Nodal Copy	`Xfer_NodalCopy_Support`
Element Copy	`Xfer_ElemCopy_Support`
Constraint-Object Copy	`Xfer_ConsObjCopy_Support`
Different Mesh Transfer	`Xfer_ParTransfer_Support`

Table 2.2. Important SIERRA Framework Version 3 Classes Used by Transfer Subsystem

Object Functionality	SIERRA Class	Notes
SIERRA Region	`Fmwk_Region`	
Geometric Decomposition	`Fmwk_GeomDecomp`	Uses Zoltan [4] for RCB decomposition
Local Search	`Fmwk_LocalSearch`	Implements point-in-box method [3]
Communication Specification	`Fmwk_CommSpec`	Used for parallel algorithms
Communication Operations	`Fmwk_CommMgr`	Used for parallel algorithms
Least-Squares Patch Projection	`Fmwk_RecoverField`	Used for the element-centroid least-squares projection transfer operator
Basis and Field Sampling for Least-Squares Patch	`Xfer_ElemCentroid Linear`	Derived from `Fmwk_RecoverField`; computes basis and element field at centroid of each element in patch
Master Element Operations	`Elem_MasterElem`	Used for different-mesh transfer operators

2.3 Intraprocedural Versus Interprocedural Transfers

Transfer operations can be performed in one of two places inside a SIERRA execution. The first is within a single procedure; this type of transfer is referred to as intraprocedural. Data is transferred between two regions that are owned by the same procedure, as shown in Fig. 2.2. For example, transfers might occur after every time step in a procedure involving the coupling of thermal and fluid analyses. The intraprocedural transfer is the most common type of transfer usage.

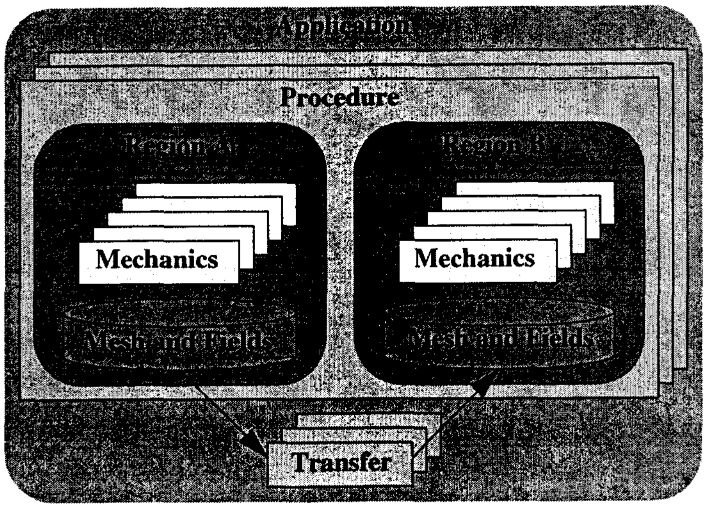

Figure 2.2. Usage of intraprocedural transfers: Data is transferred between two regions belonging to the same procedure. The application code is responsible for executing these transfers.

The second place a transfer operation can be performed in SIERRA is *between* two procedures; this type of transfer is referred to as interprocedural and can be used for a much looser coupling between the two types of physics, as illustrated in Fig. 2.3. For example, one application for interprocedural transfers is a quasi-static preload followed by a transient dynamics analysis. The quasi-static preload and transient dynamics would each have its own SIERRA procedure; and a transfer object would be used to initialize the stress tensor and displacement vector for the transient dynamics analysis, following the preload phase.

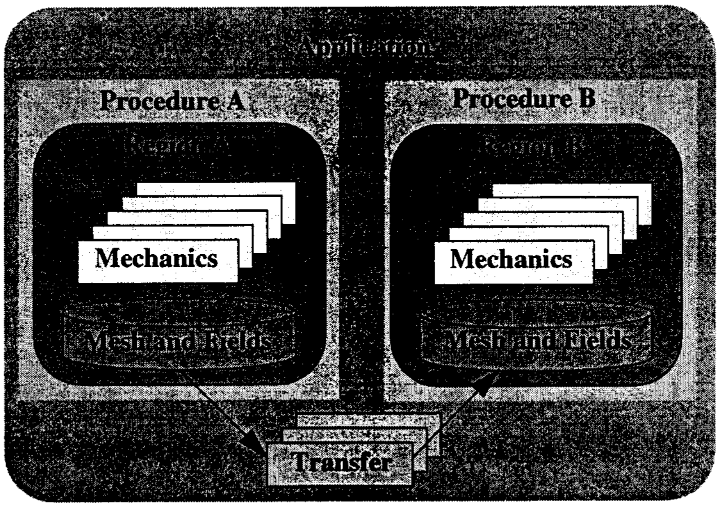

Figure 2.3. Usage of interprocedural transfers. Data is transferred between two regions belonging to different procedures. In this example, the transfers are carried out after execution of procedure A and before execution of procedure B. The execution of these transfers is handled by SIERRA.

Structurally, there is no difference between intraprocedural and interprocedural transfers. *The only difference is when these transfers can be called during execution.* This, in turn, is dictated by whether the owning procedures of the two regions involved in the transfer are the same (intraprocedural) or different (interprocedural). In a SIERRA execution, as soon as a procedure finishes executing, any interprocedural transfers involving the just-completed procedure and the subsequent procedure are carried out; then the subsequent procedure begins its execution. *The calls to these transfers (both initialization and execution) are handled by the framework.*

On the other hand, the calls to all intraprocedural transfers must be implemented by the application developer, since these calls will be made from within an application's procedure. For this reason, intraprocedural transfers require more code from the application developer. Note that interprocedural transfers potentially use less CPU time because they will probably be called less frequently.

Remark

The procedure coupling shown in Fig. 2.3 is not the only way that interprocedural transfers can be used. In certain *a posteriori* error estimators, for example, interprocedural transfers are used to transfer data between the global mesh (region A) and a local element mesh (region B). The local element mesh (equivalently, the local region) is not part of the procedure that is currently executing. Instead, a dummy procedure is created to house region B, which has the effect of making the transfers interprocedural. While this approach is mainly a technicality and has no impact on the purpose or use of the transfer, it does demonstrate the generality by which interprocedural transfers may be constructed and used. It is important to note that in this case SIERRA does *not* execute the transfers; this task is carried out inside the error estimator.

3 Transfer Operators

The transfer operators presented in this section have been implemented in SIERRA Framework Version 3. These operators are constructed incrementally by an application, based on the following general procedure:

1. Construct an (almost) empty transfer object by supplying the proper SIERRA support object corresponding to the desired transfer operator. The support objects are singletons and are constructed in the application's `main` routine.

2. Fill in the required transfer data, such as the send and receive regions, mechanics and mechanics instances (if any), etc. Note that this step cannot be done until the regions and mechanics objects have been constructed. See `Xfer_Transfer.h` for the API.

3. Construct the transfer registrars and provide the names and states of the send and receive variables. If the variables are nested, also provide the number of nesting levels (currently only the element copy operator supports nested variables).

4. Supply any required and, if desired, optional transfer control data. Some transfer operators require more control data than other transfer operators.

5. Commit the transfer object.

Note that the different-mesh transfer operators require an `initialize` method in addition to an execute method (called `transfer_state`), while the same-mesh transfer operators (nodal copy, element copy and constraint-object copy) require only the execute method.

The mechanics and mechanics instance information mentioned in step 2 warrants further discussion. One of the purposes of the mechanics and mechanics instance specifications is to define a mesh extent for the transfer operator. That is, this information tells the transfer operator what mesh objects (elements, faces, edges, nodes, or constraint objects) are involved in the transfer, in both the send and receive regions. In general, the mesh extent is determined as follows. If no mechanics is specified, then the mesh extent is the entire region. If a mechanics is specified without a mechanics instance, then the mesh extent is determined by the equivalence-use associated with the mechanics. If both a mechanics *and* a mechanics instance are specified, then the mesh extent is defined by *both* the equivalence-use associated with the mechanics *and* the mechanics instance.

The second purpose of the mechanics specification is to identify the master element to be used by the transfer operator. (Note that the Nodal Copy and Constraint-Object Copy transfer operators do not require a master element.) If a mechanics is specified, then a unique master element must be obtainable from the mechanics context and the variable to be transferred. If no mechanics is specified, then it is a requirement that only one master element family exist in the region (e.g., a hexahedral family consisting of a fully-integrated linear hexahedron and a fully-integrated quadrilateral hexahedron). For the transfer search step (in which the transfer variables are not used and, therefore, a unique master element is not determinable), the "richest" master element in the master element family is used by default (the quadrilateral hexahedron in the previous example). Remarks on the limitations of using mechanics and mechanics instances in the transfer operator API are given in Section 5.

A description of each transfer operator, including assumptions and restrictions for its usage, follows.

3.1 Nodal Copy

The SIERRA support class for the nodal copy transfer operator is `Xfer_NodalCopy_Support`. The transfer operator "copies" nodal fields from the send region onto the corresponding nodes of the receive region. This transfer is very restrictive in that the send and receive regions must have exactly the same meshes with the same parallel decompositions. This restriction implies that the global IDs of the nodes must be exactly the same in each mesh. On the other hand, this transfer is fully parallel without requiring a single communication operation. Consequently, the nodal copy transfer operator is very simple and inexpensive, and should be used whenever it is known that the send and receive regions have exactly the same meshes.

The nodal copy transfer operator can be used for the entire volume mesh, i.e., every node in the mesh, or on a subset of the mesh, including a surface or set of surfaces. The subsets are defined in terms of mesh extents that are associated with SIERRA mechanics or SIERRA mechanics instances, as discussed in the previous section.

Remark

There is a special version of the nodal copy transfer operator where the user may *hardwire* the association of a receive node with a send node. Here, it is not required that the global IDs of the nodes or the parallel decompositions of the send and receive meshes be identical. The association of the receive nodes with the send nodes is done by providing the transfer operator with a CommSpec object (i.e., a parallel communication specification; see Section 4 and Ref. 1). The actual transfer of data, therefore, involves communication as determined by the CommSpec object. The intended use of this version of the nodal copy transfer operator is for transferring data between like mesh-subsets that exist in different global meshes. For example, element block A (with identical elements, nodes, and connectivities) might exist in both the send and receive meshes. However, if there exist other element blocks in either mesh, the elements and nodes in block A in the send mesh might have different global IDs and reside on different processors than the corresponding elements and nodes in block A in the receive mesh. A CommSpec can then be constructed that hardwires the association of the corresponding block A nodes in the send and receive meshes. The nodal copy transfer can then proceed without requiring a geometric search (such a search would be required, for example, if this transfer was performed with the nodal interpolation transfer operator; see Section 3.4). An optional control data variable can be specified which would reverse the direction of the communication, so that the reverse transfer (i.e., receive mesh to send mesh) could be carried out without requiring the instantiation of a new transfer operator.

Usage and control data for the nodal copy transfer operator are highlighted below.

- A call to Xfer_NodalCopy_Support::self() must be supplied in the application's main routine.
- Optional control data:
  - CONVERSE <Int>dummy_value (This variable is only used for the special version of the nodal copy transfer operator described in the remark above. If the variable is registered, then the communication path is reversed. The default is not-registered; dummy_value is ignored.)
- There is no initialize method for this transfer operator.

3.2 Element Copy

The SIERRA support class for the element copy transfer operator is Xfer_ElemCopy_Support. The transfer operator "copies" element fields from the send region onto the corresponding elements of the receive region. Like the nodal copy transfer operator, the element copy transfer operator is very restrictive. The send and receive regions must have exactly the same meshes with the same parallel decompositions. This restriction implies that the global IDs of the elements must be exactly the same in each mesh. On the other hand, this transfer is fully parallel without requiring a single communication operation. Consequently, the element copy transfer operator is very simple and inexpensive, and should be used whenever it is known that the send and receive regions have exactly the same meshes.

An additional restriction is that the master elements of the corresponding elements in the send and receive regions must be identical, *except* when the element variable only has one coefficient, i.e., the variable is constant on the element. In that case there is no issue related to data associativity within the element, such as might happen if the variable was associated with integration points but the integration points were not the same in the two master elements being used.

The element copy transfer operator does support nested element variables. The number of nesting levels for each variable must be specified in the transfer registrar. The default number of levels is one (i.e., no nesting).

Usage of the element copy transfer operator requires that the SIERRA element mechanics be supplied, i.e., provided during construction of the transfer object. This requirement is necessary so that the correct equivalence-use [1] can be obtained for the element. Alternatively, one may specify a list of input/output instance names, i.e., element block names. In this case the transfer object automatically finds the correct SIERRA element mechanics.

Usage and control data for the element copy transfer operator are highlighted below.

- A call to Xfer_ElemCopy_Support::self() must be supplied in the application's main routine.
- There is no control data associated with this transfer operator.
- There is no initialize method for this transfer operator.

3.3 Constraint-Object Copy

The SIERRA support class for the constraint-object copy transfer operator is `Xfer_ConsObjCopy_Support`. The transfer operator "copies" constraint-object fields from the send region onto the corresponding constraint objects of the receive region. This transfer is very restrictive in that the send and receive regions must have exactly the same meshes with the same parallel decompositions. This restriction implies that the global IDs of the constraint object must be exactly the same in each mesh. On the other hand, this transfer is fully parallel without requiring a single communication operation. Consequently, the constraint-object copy transfer operator is very simple and inexpensive.

The constraint-object copy transfer operator can be used for the entire volume mesh, i.e., every constraint in the mesh, or on a subset of the mesh, including a surface or set of surfaces. The subsets are defined in terms of mesh extents that are associated with SIERRA mechanics or SIERRA mechanics instances.

A constraint-object field might be, for example, a force or a flux associated with a node-face pair that is constrained in some way. Generally, transferring such a field between different meshes is not desirable. In such cases it is often required (for accuracy in enforcing the constraint) that the constraint force or flux be recomputed on the other mesh. For this reason, a different-mesh version of the constraint-object copy transfer operator has not been developed. The constraint-object copy transfer operator is simply a means of avoiding the costly recomputation of the constraint whenever the two meshes are exactly the same.

A typical use of the constraint-object copy transfer operator is as an *interprocedural* transfer. It may be the case that the constraint objects themselves have not yet been constructed in the receive region at the time the interprocedural transfer is invoked. The reason is that constraint objects are typically not read from an input mesh file; instead, these objects are constructed during execution (in the send region). To accommodate this, the transfer operator will, if necessary, construct the receive-region constraint objects during its execution. The use of this functionality is currently very limited. A mechanics instance must be specified, and the instance names in the send and receive regions must be identical. Furthermore, the constraint-object construction has only been implemented in serial. A parallel implementation can be added if necessary.

Usage and control data for the constraint-object copy transfer operator are highlighted below.

- A call to `Xfer_ConsObjCopy_Support::self()` must be supplied in the application's `main` routine.
- There is no control data associated with this transfer operator.
- There is no `initialize` method for this transfer operator.

3.4 Nodal Interpolation

The SIERRA support class for the nodal interpolation algorithm is `Xfer_ParTransfer_Support`. The transfer operator interpolates nodal fields from the send region onto the nodes of the receive region. The interpolation is local within an element and is carried out using the finite-element shape functions. Unlike the nodal copy transfer operator, the nodal interpolation transfer operator is completely general in that it can handle two regions with independently generated meshes. The meshes can have different parallel decompositions and are not required to have conformal boundaries. For example, if the two meshes discretize a curved domain boundary in different ways, each mesh will have a number of nodes that do not lie inside the volume of the other mesh. The operator is set up to allow for different ways of handling this case. Methods available in SIERRA Framework Version 3 are *extrapolate* and *ignore*. For the extrapolate option, the receive nodes lying outside the send region are paired with the geometrically closest element in the send region. The field values for these nodes are then computed by extrapolation using the shape functions of the paired element. A user-subroutine capability for handling the "nodes outside region" is planned for the future.

The nodal interpolation transfer operator uses parallel algorithm 1, described in Section 4.2. The geometric search tolerance must be supplied by the application developer through control data. The geometric search consists of two parts—the *parallel* search and the *local* search. The parallel search uses RCB (Recursive Coordinate Bisection) to define a new decomposition for handling the geometric misalignment of the two meshes among the processors. The RCB implementation is supplied by Zoltan [4]. The local search uses the point-in-box method [3] for quickly finding all the receive nodes that lie inside each of the send elements.

The transfer operator can be used for either *volume-to-volume* transfers or *surface-to-surface* transfers. The search for the volume transfers is basically the same as that for the surface transfers. In the former case, the search uses volume master elements and returns element/node pairs (send-region elements paired with receive-region nodes). In the latter case, the search uses surface master elements and returns face/node pairs (send-region faces paired with receive-region nodes). In two dimensions, the surface-transfer search would return edge/node pairs; however, this capability has not been tested.

It is important to note that nodal variables are usually not associated with a master element. The nodal interpolation transfer operator, however, requires that a master element be registered in the send region. Moreover, it is required that the master element be associated with the nodal variables being transferred. This association is done, for example, through the assignment of a given mechanics context (see Ref. 1) to both the nodal variable and the master element. For the purpose of the search, if multiple master elements (of the same topological family) are registered for a given element, then the topologically richest element is used. For example, if the transfer operator detects that both an 8-node linear hexahedron and a 27-node quadratic hexahedron are registered (e.g., for a mixed-element formulation), the search would use the 27-node hexahedron for determining whether a receive node lies inside it.

New element types can easily be added by supplying the required `is_in_element` and `interpolate_point` master element methods. The `interpolate_point` method uses the nodal shape functions to compute the value of the nodal field at a point, given that point's parametric coordinates with respect to the element. The `is_in_element` method determines whether a point in space lies geometrically inside or outside the element. This method also computes the parametric coordinates. For nonconforming discretizations on a surface, the surface transfers would first need to project the receive node onto the send surface, then use the face (or edge in two dimensions) master element to perform the interpolation. This projection step is assumed to be part of the face or edge `is_in_element` implementation.

Surface transfers involving noncoplanar surfaces require the additional control-data specification of the *surface gap tolerance*. The surface gap tolerance is used in the definition of the local search box (used in the local search) for a given send face or edge. The local search box is initially set to be the minimum bounding box of the face or edge; this bounding box is always aligned with the coordinate axes. The bounding box is then expanded in the +/- *(x,y,z)* directions (+/- *(x,y)* directions in two dimensions) by the distance equal to the surface gap tolerance. This expansion ensures that the dimension of the local search box in the direction normal to the face or edge is *at least* at large as the tolerance. Receive nodes lying outside the search box (and, therefore, outside the surface gap tolerance) will not be found by the search algorithm.

The surface gap tolerance also affects the parallel search because the same expanded search box is used to determine the extent of *ghosting* of the send faces or edges. Ghosting is required to guarantee parallel consistency and robustness of the algorithm, i.e., to ensure that a given receive node is always paired with the same send face or edge regardless of the number of processors. In general, a larger surface gap tolerance leads to increased ghosting, resulting in more communication and larger memory use. Therefore, one should not specify a surface gap tolerance that is unnecessarily large. Recall, however, that a tolerance that is too low may cause failure of the search algorithm because it could result in receive nodes that are never found. For these reasons, the selection of the surface gap tolerance should be done with care.

Remark

It is recommended that the *ignore* option for handling the "nodes outside region" *not* be used for surface transfers, since nodes lying outside the surface gap tolerance might incorrectly be ignored.

Usage and control data for the nodal interpolation transfer operator are highlighted below.

- A call to `Xfer_ParTransfer_Support::self()` must be supplied in the application's `main` routine.
- Specification of the nodal interpolation transfer operator is done by setting the TRANSFER_TYPE control data variable:
  - TRANSFER_TYPE <enumeration>type, where type = Xfer_Transfer::NODAL_INTERP
- Other required control data:
  - SEND_MODEL_COORD_NAME <String>name
  - SEND_MODEL_COORD_STATE <String>name
  - RECV_MODEL_COORD_NAME <String>name
  - RECV_MODEL_COORD_STATE <String>name
  - GEOM_TOL <Real>search_tolerance

- OBJECTS_OUTSIDE_REGION <String>name (EXTRAPOLATE and IGNORE are the only valid options.)

- Optional control data:
  - SEND_REGION_OBJECT_TYPE <String>object_type (Valid values are ELEMENT, FACE, or EDGE; supplying this information simplifies and speeds up the transfer operator.)
  - SURFACE_GAP_TOLERANCE <Real>gap_tolerance (The default value is gap_tolerance = 0.0. A nonzero value is recommended for surface transfers involving nonplanar surfaces or surfaces with any gaps between them.)
  - XFER_DEBUG_PRINT <Int>print_flag (A nonzero value turns debug printing on. The default value is print_flag = 0.)
  - XFER_TIMERS <Int>timer_flag (timer_flag = 1 turns on transfer timing information, which is printed to the log file during execution. The default value is timer_flag = 0, which turns off the timers).
- There is an `initialize` method for this transfer operator. For intraprocedural transfers, the call to the `initialize` method must be made by the application. For interprocedural transfers used between two sequenced procedures, this call is handled by the SIERRA framework (see Section 2.3 for more a more detailed discussion).

3.5 Element-Centroid Least-Squares Projection

The SIERRA support class for this transfer operator is `Xfer_ParTransfer_Support`. The transfer operator computes element fields in the receive region using trilinear polynomials constructed from elements in the send region. The use of this transfer is restricted to piecewise-constant element fields only! Such fields can be represented on each element by a single value associated with the element centroid. The trilinear interpolating polynomials are of the form

$$\phi(x, y, z) = a_0 + a_1 x + a_2 y + a_3 z + a_4 xy + a_5 yz + a_6 xz + a_7 xyz \quad (1)$$

The coefficients $a_0, \ldots, a_7$ are computed for each send-region element, and for each scalar component of the field being transferred, using a least-squares patch projection. The patches are element-centered; i.e., each patch consists of the union of the node-centered patches of each of the element's nodes. If $(\bar{x}, \bar{y}, \bar{z})$ denotes the centroid of a receive-region element, the transferred value for a scalar field (or scalar component of a vector or tensor field), ϕ_c, is computed as

$$\phi_c = a_0 + a_1 \bar{x} + a_2 \bar{y} + a_3 \bar{z} + a_4 \bar{x}\bar{y} + a_5 \bar{y}\bar{z} + a_6 \bar{x}\bar{z} + a_7 \bar{x}\bar{y}\bar{z} \quad (2)$$

The least-squares patch projection involves solving a linear system of the form

$$\mathbf{A}^{\mathbf{T}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathbf{T}}\mathbf{b} \quad (3)$$

where

$$\mathbf{A} = \begin{bmatrix} 1\ x_1\ y_1\ z_1\ x_1y_1\ y_1z_1\ x_1z_1\ x_1y_1z_1 \\ \ldots \\ 1\ x_n\ y_n\ z_n\ x_ny_n\ y_nz_n\ x_nz_n\ x_ny_nz_n \end{bmatrix} \tag{4}$$

$$\mathbf{x} = \begin{bmatrix} a_0\ a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7 \end{bmatrix}^{\mathrm{T}} \tag{5}$$

$$\mathbf{b} = \begin{bmatrix} \phi_1\ \ldots\ \phi_n \end{bmatrix}^{\mathrm{T}} \tag{6}$$

and where n is the number of sampling points in the patch. Because we are restricted to piecewise-constant element fields, n is also equal to the number of elements in the patch. The implementation of the least-squares projection[1] is in the SIERRA class `Fmwk_RecoverField` (see `Fmwk_RecoverField.h`). This implementation was put in the framework, i.e., not in the transfer subsystem, because the projection algorithm is also to be used for purposes other than transfers, such as the Zienkiewicz-Zhu error estimator [5] and the element prolongation algorithm in codes such as *adagio* [6]. `Fmwk_RecoverField` is an abstract base class requiring derivation for each particular application. The derived class must implement a virtual method that computes the basis vector, i.e., the rows of $\mathbf{A}$, and the element field, i.e., the rows of $\mathbf{b}$, for each sampling point in the patch. This method is implemented for the *element-centroid least-squares projection* transfer operator in the class `Xfer_ElemCentroidLinear`.

Of concern here is the solvability of (4). If $n < 8$, indicating that not enough elements are in the patch, $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ will not possess full rank and (4) will not be solvable. Rank deficiency can also occur for $n \geq 8$ if the sampling points are positioned in degenerate ways, such as if all the points lie in a plane. Figure 3.1 illustrates a solvable patch and an unsolvable patch in a simple two-dimensional mesh of a "T" shape using quadrilateral elements. The elements at or near the top ends of the "T" have insufficient patches for solvability; the patches for the other elements are sufficient. (Note that the two-dimensional analogy of (1) is $\phi(x, y) = a_0 + a_1x + a_2y + a_3xy$; therefore $n \geq 4$ for solvability. The extension to three dimensions should be clear.)

1. Much of the least-squares projection transfer operator was originally developed by Kevin Copps.

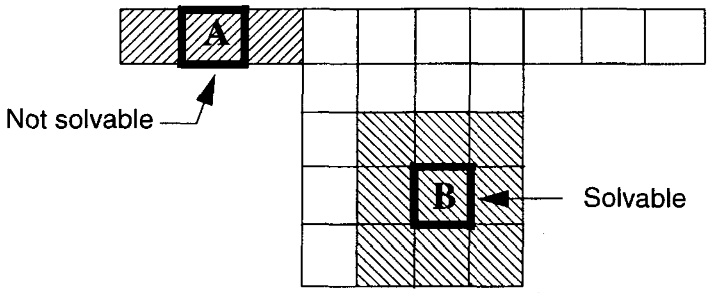

Figure 3.1. Illustration of a solvable patch and an unsolvable patch. Element A ($n = 3$) does not have enough neighbors to form a full-rank system for the least-squares projection, while element B ($n = 9$) has a large-enough patch to ensure solvability (n is the size of the patch).

To handle the rank-deficient systems, the least-squares implementation computes the condition number of $\mathbf{A}^T\mathbf{A}$, then systematically removes the linearly dependent columns and rows that cause the rank-deficiency[2]. This leads to a smaller, full-rank system; the resulting projection is with respect to a reduced basis. For example, if in three dimensions all the sampling points lied in the *xy* plane, columns 4, 6, 7 and 8 would be linearly dependent and thus removed from the system, leading to the reduced basis $(1, x, y, xy)$. The computed receive-region element-centroid value would then be $\phi_c = a_0 + a_1\bar{x} + a_2\bar{y} + a_4\bar{x}\bar{y}$. In the limit of a patch consisting of a single element, all rows and columns except the first would be eliminated, leading to the (correct) projection $\phi_c = a_0$, where a_0 would be the (send-region) element field value.

If computational efficiency of the least-squares projection is a concern, then the condition number check and the subsequent removal of any linearly dependent columns and rows can be skipped. This is done by specifying the relative diagonal tolerance to be 0.0. A nonzero tolerance (where $0.0 < \text{tol} \leq 1.0$) triggers the condition number check. The tolerance scales the largest diagonal entry in $\mathbf{A}^T\mathbf{A}$; any diagonal entries smaller than this scaled value are removed from the system as described above. It should be noted that although skipping the condition number check could speed up the algorithm, it could also cause the algorithm to fail if the system is ill-conditioned. Therefore, one must use caution when requesting that the condition number be skipped.

Unlike the element copy transfer operator, the element-centroid least-squares projection transfer operator is completely general in that it can handle two regions with independently generated meshes. The meshes can have different parallel decompositions and are not required to have conformal boundaries. For example, if the two meshes discretize a curved domain boundary in different ways, each mesh may have a number of elements with centroids that do not lie inside the volume of the other mesh. The transfer operator is set up to allow for different ways of handling this case. Methods available in SIERRA Framework Version 3 are *extrapolate* and *ignore*. For the extrapolate option, the receive elements lying outside the send region are paired with the

2. This algorithm was developed by Jason Hales.

geometrically closest element in the send region. The field values for these elements are then computed by extrapolation using the least-squares interpolating polynomial constructed for the paired element. A user-subroutine capability for handling the "elements outside region" is planned for the future.

The projection algorithm, obviously, is not local within an element; therefore, ghost elements (and their nodes) must be created to form the patches for elements on processor boundaries. This is illustrated in Fig. 3.2 for the "T" mesh shown previously in Fig. 3.1. Here, the mesh is distributed as shown among two processors. The mesh-object ghosting procedure creates read-only temporary copies of each processor-boundary element (along with their nodes) on the adjacent processor. In this example, three of the ghosted elements on processor 1 would be used in the patch for element B. The ghosting is a service provided by the framework (see `Fmwk_MeshManufacture.h`) and used by the transfer operator.

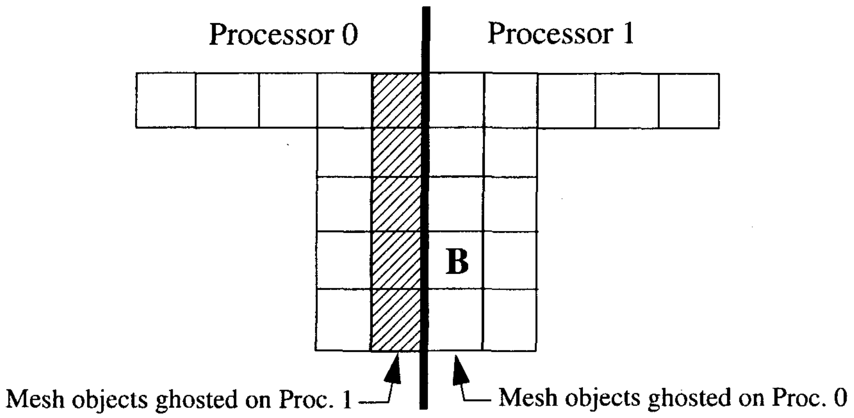
Processor 0
Processor 1
B
Mesh objects ghosted on Proc. 1
Mesh objects ghosted on Proc. 0

Figure 3.2. Ghosting of mesh objects. Such ghosting is necessary to form the patches for the least-squares projection. In this example, element B has three ghost elements in its patch.

The element-centroid least-squares projection transfer operator uses parallel algorithm 1, described in Section 4.2. The geometric search tolerance must be supplied by the application developer through control data. The geometric search consists of two parts—the *parallel* search and the *local* search. The parallel search uses RCB to define a new decomposition for handling the geometric misalignment of the two meshes between the processors. The RCB implementation is supplied by Zoltan [4]. The local search uses the point-in-box method [3] for quickly finding all the receive-element centroids that lie inside each of the send elements.

The transfer operator can be used for either *volume-to-volume* transfers or *surface-to-surface* transfers. In the latter case, the transfers would involve piecewise-constant face or edge variables in three and two dimensions, respectively. ***As of the release of SIERRA Framework Version 3.01, the surface transfers have not been tested!*** The search for the volume transfers is basically the same as that for the surface transfers. For volume transfers, the search uses volume master elements and returns element/element pairs (send-region elements paired with receive-region elements). For surface transfers, the search uses surface master elements and returns face/face pairs (send-region faces paired with receive-region faces). In two dimensions, the surface-transfer search would return edge/edge pairs.

It is important to note that even though master elements are not used for the computation of the field value, they are still required for the search algorithm! Therefore, it is necessary that the send-region element field be registered with a master element usage (see Ref. 1). The master element is required to have an implementation of the `is_in_element` virtual method (see `Elem_MasterElem.h`). This method determines whether a point in space (the receive-element centroid) lies geometrically inside or outside the (send-region) element. The method also computes the parametric coordinates of the point with respect to the element. For nonconforming discretizations on a surface, the surface transfers would first need to project the point onto the send surface, then use the face (or edge in two dimensions) master element to perform the in/out test. This projection step is assumed to be part of the face or edge `is_in_element` implementation. New element types can easily be added by supplying the required `is_in_element` master element method.

For the purpose of the search, if multiple master elements (of the same topological family) are registered for a given element, then the topologically richest element is used. For example, if the transfer operator detects that both an 8-node linear hexahedron and a 27-node quadratic hexahedron are registered, e.g., for a mixed-element formulation, the search would use the 27-node hexahedron for determining whether a receive-element centroid lies inside it.

Surface transfers involving noncoplanar surfaces require the additional control-data specification of the *surface gap tolerance*. The surface gap tolerance is used in the definition of the local search box (used in the local search) for a given send face or edge. The local search box is initially set to be the minimum bounding box of the face or edge; this bounding box is always aligned with the coordinate axes. The bounding box is then expanded in the *+/- (x,y,z)* directions *(+/- (x,y)* directions in two dimensions) by the distance equal to the surface gap tolerance. This expansion ensures that the dimension of the local search box in the direction normal to the face or edge is *at least* at large as the tolerance. Receive-face centroids (or receive-edge centroids in two dimensions) lying outside the search box (and, therefore, outside the surface gap tolerance) will not be found by the search algorithm.

The surface gap tolerance also affects the parallel search because the same expanded search box is used to determine the extent of *ghosting* of the send faces or edges. Ghosting is required to guarantee parallel consistency and robustness of the algorithm, i.e., to ensure that a given receive face (or receive edge) is always paired with the same send face (or edge) regardless of the number of processors. In general, a larger surface gap tolerance leads to increased ghosting, resulting in more communication and larger memory use. Therefore, one should not specify a surface gap tolerance that is unnecessarily large. Recall, however, that a tolerance that is too low may cause failure of the search algorithm since it could result in receive nodes that are never found. For these reasons, the selection of the surface gap tolerance should be done with care. Note that since the least-squares patch recovery requires ghosting anyway, this is not as critical of an issue as it is for the nodal interpolation transfer operator (see Section 3.4).

Remark

It is recommended that the *ignore* option for handling the "faces (or edges in two dimensions) outside region" *not* be used for surface transfers, since mesh objects lying outside the surface gap tolerance might incorrectly be ignored.

Usage and control data for the element-centroid least-squares projection transfer operator are highlighted below.

- A call to `Xfer_ParTransfer_Support::self()` must be supplied in the application's `main` routine.
- Specification of the element-centroid least-squares projection transfer operator is done by setting the TRANSFER_TYPE control data variable:
  - TRANSFER_TYPE <enumeration>type, where type = Xfer_Transfer::ELEM_CENTROID_LINEAR
- Other required control data:
  - SEND_MODEL_COORD_NAME <String>name
  - SEND_MODEL_COORD_STATE <String>name
  - RECV_MODEL_COORD_NAME <String>name
  - RECV_MODEL_COORD_STATE <String>name
  - GEOM_TOL <Real>search_tolerance
  - OBJECTS_OUTSIDE_REGION <String>name (EXTRAPOLATE or IGNORE are the only valid options.)
- Optional control data:
  - SEND_REGION_OBJECT_TYPE <String>object_type (Valid values are ELEMENT, FACE, or EDGE; supplying this information simplifies and speeds up the transfer operator.)
  - SURFACE_GAP_TOLERANCE <Real>gap_tolerance (The default value is gap_tolerance = 0.0. A nonzero value is recommended for surface transfers involving nonplanar surfaces and surfaces with any gaps between them.)
  - XFER_LS_PROJECTION_DIAGONAL_TOL <Real>diagonal_tolerance (Valid values are $0.0 \le \text{diagonal\_tolerance} \le 1.0$. If $\text{diagonal\_tolerance} = 0.0$, the condition number computation in the least-squares projection algorithm is skipped. The default value is $\text{diagonal\_tolerance} = 1.0 \times 10^{-10}$.)
  - XFER_DEBUG_PRINT <Int>print_flag (A nonzero value turns debug printing on. The default value is print_flag = 0.)
  - XFER_TIMERS <Int>timer_flag (timer_flag = 1 turns on transfer timing information, which is printed to the log file during execution. The default value is timer_flag = 0, which turns off the timers)

- There is an `initialize` method for this transfer operator. For intraprocedural transfers, the call to the `initialize` method must be made by the application. For interprocedural transfers used between two sequenced procedures, this call is handled by the SIERRA framework (see Section 2.3 for more a more detailed discussion).

4 Parallel Algorithms for Transfers

Among the most difficult and complex aspects of different-mesh SIERRA transfer operators are the parallel parts of the algorithms. The operators must account for different parallel distributions of the two meshes involved in the transfer operation. The general (and most common) case is when a processor owns a given chunk of space in one mesh and another (nonoverlapping) chunk in the other mesh. This case is common because often the two meshes are generated entirely independently of each other. Transfer operators involve intensive geometric searching. It is the responsibility of the transfer operator to geometrically align the two meshes on each processor so that the geometric searching can take place. This geometric alignment is the primary purpose of the parallel algorithm.

Different parallel algorithms are possible. Each has trade-offs involving load balance and message sizes. In SIERRA, parallel algorithms can be designed at a high level using the ideas of the *multiple concurrent domain decompositions* as well as the *parallel communication specification* (CommSpec). We describe the idea of multiple concurrent domain decompositions in the following section. This is a general capability provided by SIERRA that has applicability beyond transfers. The CommSpec is a topological relation connecting mesh objects of possibly different types (e.g., element->node) belonging to possibly different SIERRA regions. The CommSpec is, in effect, a parallel intermesh "connectivity," and can be thought of as a generalization of the standard connectivity data structure used in traditional finite element analysis. The CommSpec is described in much more detail in Ref. 1,

Two parallel algorithms for the transfer operators are described below. The better one to use depends on the specific application (see the discussion in Section 4.4). It is important to note that ***only parallel algorithm 1 is implemented in SIERRA Framework Version 3***. Implementation of both algorithms has been done outside of SIERRA and demonstrated on nonphysical model problems [7]. Ref. 7 also presents performance results for the algorithms.

4.1 Multiple Concurrent Domain Decompositions

In a distributed parallel environment, each processor owns a different piece of the problem (with small regions of overlap, or ghosting). This decomposition is usually constructed to balance the computational load while minimizing interprocessor communication, which amounts to parallel "overhead." In SIERRA, such a decomposition is referred to as the *primary decomposition*. The primary decomposition is obtained through a preprocessing step (outside of SIERRA); the decomposed mesh is subsequently read in as input.

In many situations the primary decomposition will either impede scalability or cause an algorithm to fail. For these reasons SIERRA provides a mechanism to dynamically create a *secondary* *decomposition*, whereby a copy of the mesh (or appropriate mesh subset) is created according to the requirements of the algorithm. Perhaps the algorithm operates only on a subset of the mesh, such as a surface, which may in turn reside only on a subset of the processors (in the primary decomposition). If this algorithm was executed in the primary decomposition, the remaining processors would sit idle. The secondary decomposition would dynamically balance the computational load by equally distributing the surface among all the processors.

For other algorithms such as the parallel transfer, the algorithm may *fail* in the primary decomposition (since on-processor geometric alignment of the two meshes is required). The SIERRA parallel transfer algorithms achieve geometric alignment on the processors by constructing a secondary decomposition, sometimes called a *rendezvous decomposition* [7]. This decomposition is geometrically based and can be constructed, for example, by an RCB algorithm. Geometric alignment of the two meshes, i.e., the send and receive meshes, is guaranteed because *both* meshes are reconstructed in this new decomposition. The reconstructed meshes are referred to as *secondary regions* in SIERRA (these regions are more than just meshes—they are full-fledged SIERRA regions that may also contain registered field data). The secondary regions are copies of the original, i.e., primary, regions, implying that sufficient memory must be available on each processor. However, usually only a subset of the primary regions is copied because the transfer, in general, does not need everything in the primary region. In addition, the secondary region is *temporary*; it is created for the sole purpose of the transfer, then deleted upon deletion of the transfer object.

An application (including the transfer operator) uses the following SIERRA services in conjunction with constructing secondary decompositions.

- Definition of a secondary load-balanced decomposition for the appropriate subset of mesh objects (SIERRA uses the Zoltan dynamic load-balancing library [4] for this purpose).
- Construction of a communication map, i.e., a CommSpec, between the primary decomposition and the secondary decomposition.
- Construction of a copy of the appropriate region subset in the secondary decomposition.
- Communication of field values between the primary and secondary regions.

A simple illustration of multiple concurrent domain decompositions is given in Fig. 4.1, which shows the overlay of a primary and a secondary decomposition on four processors for a two-dimensional mesh. The primary decomposition is topologically based, while the secondary region is geometrically based, making it suitable for geometric-searching algorithms. The secondary decomposition shown in the figure is typical of one generated by RCB.

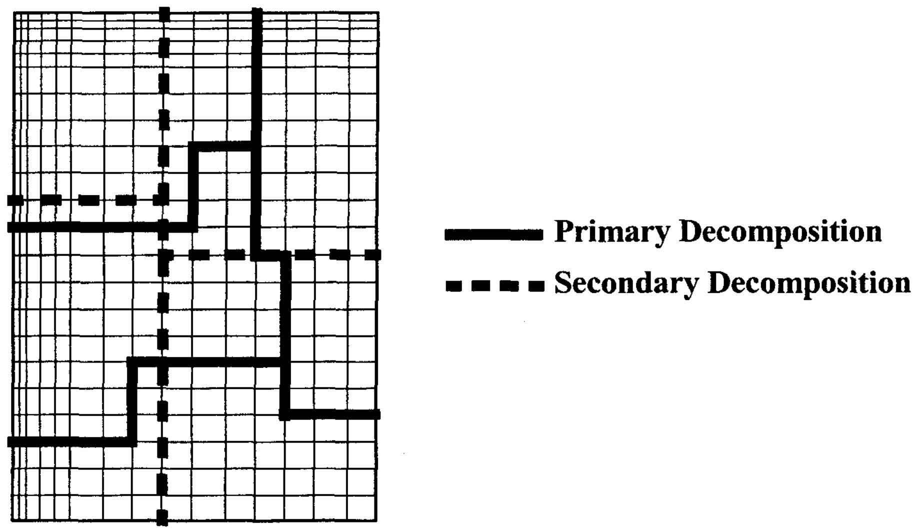

Figure 4.1. Illustration of multiple concurrent domain decompositions. The primary and secondary decompositions are overlaid on four processors for a two-dimensional mesh.

4.2 Parallel Algorithm 1

The algorithm design is shown in Fig. 4.2. We wish to transfer registered field variables from the send (primary) region to the receive (primary) region. The sequence of steps in the algorithm is described below. For concreteness, the discussion is restricted to nodal-interpolation volume transfers, i.e., the volume-to-volume transfer of nodal variables; see Section 3.4.

- Generate the secondary decomposition for the send region and the receive region. This is done using the RCB algorithm supplied by Zoltan [4]. The RCB cuts are based on the elements of the send region and the nodes of the receive region.
- Construct the send secondary region and receive secondary region. Mappings between the corresponding primary and secondary regions are stored in CommSpec objects.
- Communicate the send mesh *and* physics data from the send primary region to the send secondary region. The mesh data consists of information about the elements and nodes.
- Communicate the receive mesh data from the receive primary region to the receive secondary region. The mesh data consists of information about the nodes.
- For each node in the receive secondary region, find the element in the send secondary region that contains the node. Because we are in the secondary decomposition, this search is entirely local, i.e., on-processor; it is done using the point-in-box algorithm [3]. The node-element pairs are stored in another CommSpec object, which in this case is a "local" CommSpec. (See CommSpec C in Fig. 4.3).

- Still in the secondary decomposition, interpolate (for example, using the finite-element shape functions) the values onto each receive node using the nodal values of the send element that contains the node.
- Communicate the interpolated nodal values from the receive secondary region back to the receive primary region. This step is easily done by transposing the CommSpec connecting those two regions (CommSpec B), which reverses the communication path.

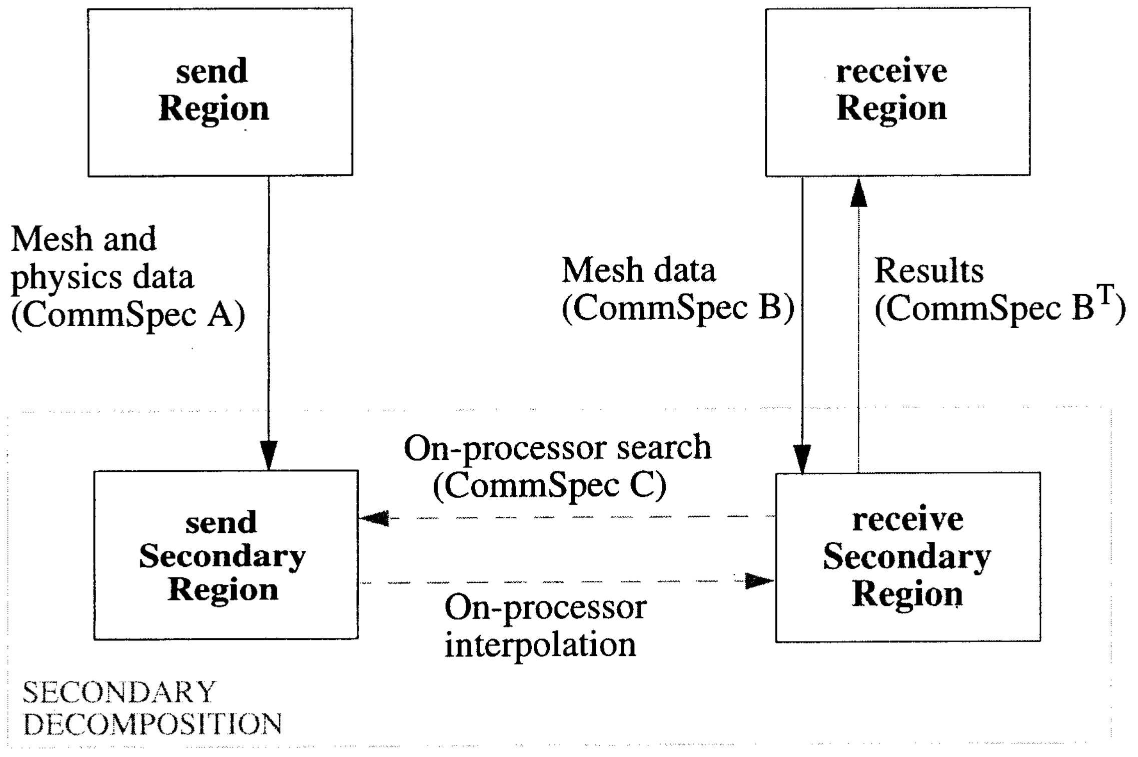

Figure 4.2. Design of parallel algorithm 1 for transfer operators.

4.3 Parallel Algorithm 2

The algorithm design is shown in Fig. 4.3. As in the above discussion, we wish to transfer registered field variables from the send (primary) region to the receive (primary) region. The sequence of steps in the algorithm is described below. Again for concreteness, the discussion is restricted to nodal-interpolation volume transfers, i.e., the volume-to-volume transfer of nodal variables; see Section 3.4.

- (Same step as parallel algorithm 1) Construct the send secondary region and receive secondary region. Mappings between the corresponding primary and secondary regions are stored in CommSpec objects.
- Communicate the send mesh data from the send primary region to the send secondary region. The mesh data consists of information about the elements and nodes.

- Communicate the receive mesh data from the receive primary region to the receive secondary region. The mesh data consists of information about the nodes.
- For each node in the receive secondary region, find the element in the send secondary region that contains the node. Because we are in the secondary decomposition, this search is entirely local, i.e., on-processor; it is done using the point-in-box algorithm [3]. The node-element pairs are stored in another CommSpec object, which in this case is a "local" CommSpec. (See CommSpec C in Fig. 4.3).
- *Redistribute* the receive secondary region to match the parallel decomposition of the send primary region. That is, communicate each receive node to the primary send processor that owns the send element containing that node. The CommSpec used for this step is obtained by composing CommSpec C with the transpose of the CommSpec that connects the send primary region to the send secondary region. (See CommSpec D in Fig. 4.3.)
- Communicate the receive/send search data from the receive secondary region to the redistributed receive secondary region. For example, this information might consist of the ID of the send element that contains each receive node.
- Now in the redistributed receive region, interpolate (for example, using the finite-element shape functions) the values onto each receive node using the nodal values of the send element that contains the node.
- Communicate the interpolated nodal values from the redistributed receive secondary region back to the receive primary region. This is done using a new CommSpec created by composing the transpose of CommSpec D (see Fig. 4.3) with the transpose of the CommSpec that connects the receive primary region to the receive secondary region (CommSpec B).

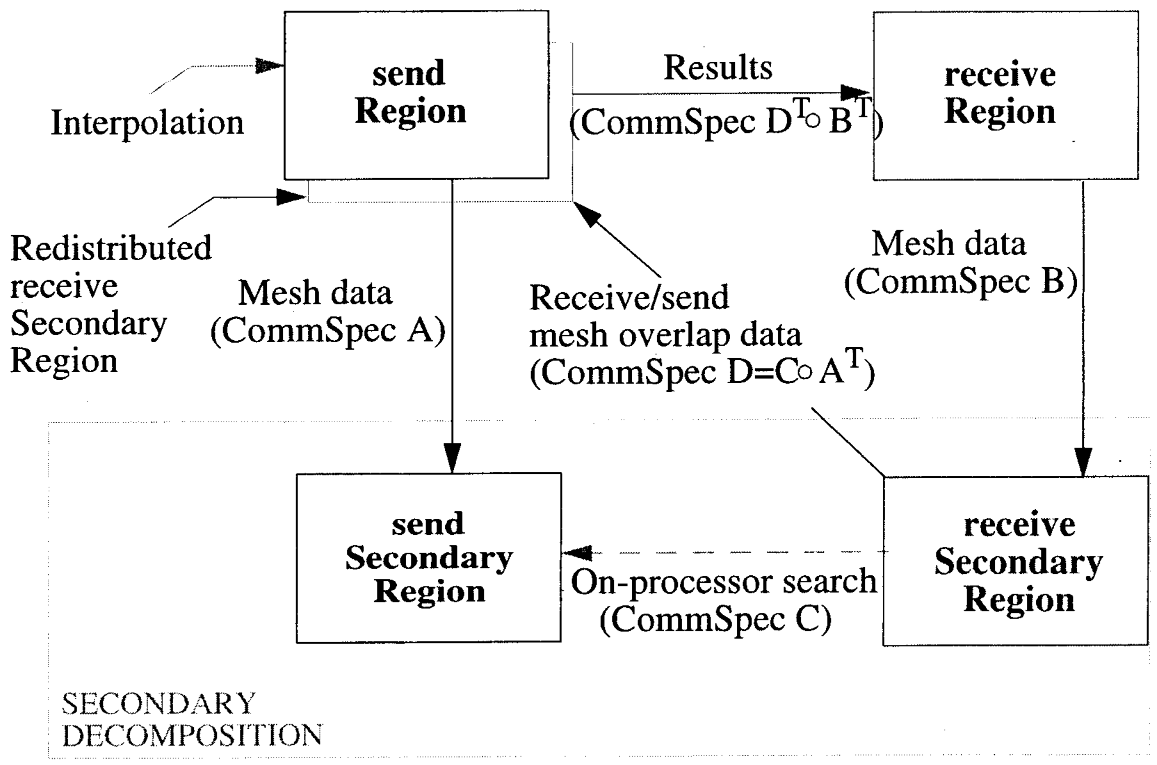

Figure 4.3. Design of parallel algorithm 2 for transfer operators.

4.4 Trade-offs Between Parallel Algorithms 1 and 2

The main difference between the two parallel algorithms is the decomposition in which the actual interpolation of the data is performed. In parallel algorithm 1, the interpolation is performed in the secondary decomposition; in parallel algorithm 2, the interpolation is performed in the send primary decomposition.

Algorithm 2 has more communication steps than algorithm 1, but potentially smaller messages. There is also the potential (in algorithm 2) that the interpolation step will become unbalanced if too many (or few) of the receive nodes end up on a single processor in the send primary decomposition. Note that this is possible because the receive primary mesh is independent of the send primary decomposition, and disproportionately many (or few) receive nodes could geometrically intersect the volume occupied by the send elements on a given processor. In algorithm 1, this situation can be controlled if the secondary decomposition (where the interpolations are carried out) is generated using input from *both* the send and receive meshes. For the nodal-interpolation volume transfer, this implies that the RCB cuts should be generated using both the elements in the send region and the nodes in the receive region (see the first step in algorithm 1). For the element-centroid least-squares projection transfer operator, this implies that the RCB cuts should be generated using both the elements in the send region and the elements in the receive region.

Finally, algorithm 2 entails an additional copy of the receive secondary region. This is due to the redistribution step, where the receive secondary region is redistributed to match the parallel decomposition of the send primary region. However, because the secondary region does not contain any physics data, the memory requirements still might not be significant when compared to the memory requirements of algorithm 1.

Due to the above trade-offs, the choice of the best parallel algorithm is most likely problem dependent.

5 Summary and Future Work

In this paper we have presented the SIERRA transfer subsystem for SIERRA Framework Version 3. The transfer operators within this subsystem are used for transferring nodal, element and constraint-object fields from one SIERRA region to another SIERRA region in a fully distributed parallel environment. "Copy" transfer operators are available for efficiency when it is known that the send mesh and the receive mesh are *identical.* For the general (different-mesh) case, a nodal interpolation transfer operator has been implemented for nodal fields, and an element-centroid least-squares patch recovery transfer operator has been implemented for (piecewise-constant) element fields. The transfer operators can be used for volume-to-volume transfers or surface-to-surface transfers. Two different parallel designs for the transfer algorithms were presented, although only one of them has been implemented thus far in the framework.

The transfer subsystem will continue to evolve in response to requirements by SIERRA customers. Among the likely paths of this evolution are the following:

- Generalization and enhancement of the transfer API. As discussed in Section 3, the mesh extents (i.e., the sets of mesh objects) and master elements needed to execute the transfer operator are extracted from SIERRA mechanics objects, which are given as input to the transfer object. Currently this is the *only* way for the application developer to specify the mesh extents and master elements (except for the Nodal Copy and Element Copy transfer operators, where input/output instance names can alternatively be given). This interface has proven to be both overly complex and limiting. For example, it may be desirable to perform a parallel transfer operation on the set of elements that excludes ghosted elements. Since this element subset is not defined by a SIERRA mechanics object, the specification of such a mesh extent for the transfer operation is not currently possible. The master element specification (currently via a SIERRA mechanics object) has proven to be difficult because of the complexity in setting up SIERRA mechanics objects such that unique master elements can be extracted. A possible generalization of the transfer API might be to allow for *direct* specification of the mesh extents and master elements by the application developer.

- Addition of more transfer operators for piecewise-constant element fields. One possibility is the direct assignment of the send-element value to each of the receive elements whose centroid is contained within it. Thus, the least-squares patch recovery and interpolation steps would be skipped. Besides being potentially much less costly, this operator could be used in conjunction with state updates of the (receive-region) element fields by the application code following the execution of the transfer.

- Implementation of transfer operators for higher-order, i.e., linear and above, element fields. Such a field might be an element variable for a fully integrated element (with more than one integration point) whose values are stored at the integration points. This capability might be needed, for example, for surface transfers of face variables. One issue is the expansion of the search to include all points associated with the element field, not just the element centroid, i.e., find the send-element *patch* that contains the receive-element.

- Encapsulation of the parallel search capability. This enhancement would allow an application to create and re-use transfer "search" objects independently of any transfer interpolation algorithm. The search information could then be used for purposes other than transfers. Currently, parallel algorithm 1 stores the search information only in the secondary region. The encapsulated search capability would require that the search information be communicated back to the primary decompositions. Parallel algorithm 2 is a step in this direction; however, that algorithm only sends the search information back to the *send* primary region. It may also be desirable to communicate this information back to the *receive* primary region.

- Performance enhancements of the transfer operators. The memory and CPU requirements of the transfer operators will become more critical in simulations involving changing mesh topologies, such as adaptive simulations. The reason is that the transfer operator will have to be "re-initialized" every time the mesh changes, e.g., via adaptive mesh refinement or unrefinement. Re-initialization is a costly operation because it performs the geometric search. Other parts of the transfer operators are also candidates for performance improvements, such as the least-squares patch recovery used for the element fields. This recovery is done during every execution of the transfer (not just whenever the mesh topology changes) because the values of the fields will potentially have changed. The intensive interprocessor communication required in the different-mesh transfers may also be a CPU bottleneck. Memory and CPU performance of the transfer operators has not been a priority to date, but will likely become one in the future.

- Specialization of the transfer operators. For certain applications, information may be available to allow for algorithmic shortcuts. In certain *a posteriori* error estimators, for example, transfers are used to move fields from a global coarse mesh to a local mesh that is generated by refining a single coarse-mesh element. In this case the entire search step could be skipped, given that there is just a single send-mesh element involved in the transfer.

References

1. H. C. Edwards. *SIERRA Framework Version 3: Core Services Theory and Design.* SAND2002-3616. Albuquerque, NM: Sandia National Laboratories, 2002.

2. G. W. Wellman. *MAPVAR - A Computer Program to Transfer Solution Data Between Finite Element Meshes.* SAND99-0466. Albuquerque, NM: Sandia National Laboratories, 1999.

3. M. E. Davis, M. W. Heinstein, S. W. Attaway, and J. W. Swegle. *Optimizing the Point-in-Box Search Algorithm for the Cray Y-MP Supercomputer.* SAND93-0933. Albuquerque, NM: Sandia National Laboratories, 1998.

4. K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. *Zoltan: A Dynamic Load-Balancing Library for Parallel Applications - User's Guide.* SAND99-1377. Albuquerque, NM: Sandia National Laboratories, 1999.

5. O. C. Zienkiewicz and J. Z. Zhu, "A Simple Error Estimator in the Finite Element Method," *International Journal for Numerical Methods in Engineering* 24 (1987): 337–357.

6. J. A. Mitchell, A. S. Gullerud, W. M. Scherzinger, R. Koteras, and V. L. Porter, "Adagio: Non-Linear Quasi-Static Structural Response Using the SIERRA Framework." In *Proceedings of the First MIT Conference in Computational Fluid and Structural Mechanics,* 361–364. Amsterdam: Elsevier, 2001.

7. S. Plimpton, B. Hendrickson, and J. Stewart. "A Parallel Rendezvous Algorithm for Interpolation Between Multiple Grids." Submitted to *Journal of Parallel and Distributed Computing.*

Distribution

External

Texas Institutute for Computational and Applied Mathematics
University of Texas at Austin
Austin, TX 78712
Attn: J. Tinsley Oden

Lawrence Livermore National Laboratories
P.O. Box 808
Livermore, CA 94551-0808
Attn: Evi Dube

Los Alamos National Laboratory
P.O. Box 1663, MS F652
Los Alamos, NM 87545
Attn: James S. Peery

Colorado State University
Department of Mathematics
101 Weber Building
Fort Collins, CO 80523-1874
Attn: Don Estep

Internal

1	MS 0841	9100	T. C. Bickel
1	MS 0835	9140	J. M. McGlaun
1	MS 0835	9113	S. N. Kempka
10	MS 0827	9143	J. D. Zepper
1	MS 0824	9110	A. C. Ratzel
1	MS 0421	9800	W. L. Hermina,
1	MS 0834	9114	J. E. Johannes
1	MS 0836	9115	E. S. Hertel
1	MS 0847	9120	H. S. Morgan
1	MS 0824	9130	J. L. Moya
1	MS 0828	9133	M. Pilch
1	MS 0847	9211	S. A. Mitchell
1	MS 1110	9214	D. E. Womble
1	MS 0819	9231	E. A. Boucheron
1	MS 0139	9900	M. O. Vahle

1	MS 0835	9141	S. W. Bova
1	MS 0835	9141	R. J. Cochran
1	MS 0835	9141	S. P. Domino
1	MS 0835	9141	M. W. Glass
1	MS 0835	9141	R. R. Lober
1	MS 0835	9141	A. A. Lorber
1	MS 0835	9141	P. A. Sackinger
1	MS 0835	9141	J. H. Strickland
1	MS 0835	9141	S. R. Subia
1	MS 9217	8920	C. J. Aro
1	MS 9042	8728	C. D. Moen
1	MS 0826	9113	D. R. Noble
1	MS 0826	9114	E. S. Piekos
1	MS 0834	9114	M. M. Hopkins
1	MS 0834	9114	P. K. Notz
1	MS 0825	9115	J. L. Payne
1	MS 0838	9116	R. E. Hogan
1	MS 0893	9123	G. W. Wellman
1	MS 0828	9133	K. J. Dowding
1	MS 0847	9133	W. R. Witkowski
1	MS 0316	9233	C. C. Ober
1	MS 0316	9233	T. M. Smith
1	MS 0316	9233	R. Hooper
1	MS 0847	9142	M. K. Bhardwaj
1	MS 0847	9142	M. L. Blanford
1	MS 0847	9142	A. S. Gullerud
1	MS 0835	9142	J. D. Hales
1	MS 0847	9142	M. W. Heinstein
1	MS 0847	9142	S. W. Key
1	MS 0847	9142	W. S. Klug
1	MS 0847	9142	J. R. Koteras
1	MS 0847	9142	N. K. Crane
1	MS 0847	9142	J. A. Mitchell
1	MS 0835	9142	K. H. Pierson
1	MS 0847	9142	V. L. Porter
1	MS 0847	9142	T. J. Preston
1	MS 0847	9142	G. M. Reese
1	MS 0847	9142	T. F. Walsh
1	MS 0807	9338	B. H. Cole
1	MS 0847	9142	K. F. Alvin
1	MS 9217	9214	M. F. Adams

1	MS 0847	9127	J. Jung
1	MS 9405	8726	R. E. Jones
1	MS 0847	9211	M. S. Eldred
1	MS 0827	9143	K. M. Aragon
1	MS 0827	9143	K. N. Belcourt
1	MS 0827	9143	D. M. Brethauer
1	MS 0827	9143	K. D. Copps
10	MS 0827	9143	H. C. Edwards
1	MS 0827	9143	C. A. Forsythe
1	MS 0827	9143	M. E. Hamilton
1	MS 0827	9143	J. R. Overfelt
1	MS 0827	9143	J. S. Rath
1	MS 0827	9143	G. D. Sjaardema
20	MS 0827	9143	J. R. Stewart
1	MS 0827	8920	A. B. Williams
1	MS 1111	9215	K. D. Devine
1	MS 0819	9231	K. H. Brown
1	MS 0819	9231	K. G. Budge
1	MS 0819	9231	S. P. Burns
1	MS 0819	9231	D. E. Carroll
1	MS 0819	9231	R. R. Drake
1	MS 0847	9226	S. J. Owen
1	MS 9018	8945-1	Central Technical Files
2	MS 0899	9616	Technical Library
1	MS 0612	9612	Review & Approval Desk for DOE/OSTI